SIANI
División de Robótica y Oceanografía Computacional

UNIVERSIDAD DE LAS PALMAS
DE GRAN CANARIA

"A Component-Based C++ Communication Middleware for an Autonomous Robotic Sailboat"

Francisco J. Santana-Jorge, Antonio C. Domínguez-Brito and Jorge Cabrera-Gámez

Instituto Universitario SIANI (www.roc.siani.es),
Departamento de Informática y Sistemas (www.dis.ulpgc.es)
Universidad de Las Palmas de Gran Canaria (www.ulpgc.es), Spain

in: Øvergård, Kjell Ivar (eds) Robotic Sailing 2017. Springer, Cham. DOI: 10.1007/978-3-319-72739-4_4

BibTeX:

```
@inproceedings{santana_jorge_et_al_2018_irsc_2017,
  author="Santana-Jorge, Francisco J.
  and Dom{\'i}nguez-Brito, Antonio C.
  and Cabrera-G{\'a}mez, Jorge",
  editor="{\O}verg{\aa}rd, Kjell Ivar",
  title="A Component-Based C++ Communication Middleware for an Autonomous Robotic Sailboat",
  booktitle="Robotic Sailing 2017",
  year="2018",
  publisher="Springer International Publishing",
  address="Cham",
  pages="39--54",
  abstract="The new C++ standard, C++11, and its upgrade, C++14, introduces new advances and features which make more affordable
  isbn="978-3-319-72739-4",
  doi="10.1007/978-3-319-72739-4_4"
}
```

Copyright © 2018, Springer International Publishing AG (www.springer.com)

document created on: 16th March 2020
created from file: irsc_2017_camera_ready.tex
cover page automatically created with CoverPage.sty
(available at your favourite CTAN mirror)

UNIVERSIDAD DE LAS PALMAS
DE GRAN CANARIA

A component-based C++ communication middleware for an autonomous robotic sailboat

Francisco J. Santana-Jorge[†], Antonio C. Domínguez-Brito[†‡§] and Jorge Cabrera-Gámez[†‡]

Abstract

The new C++ standard, C++11, and its upgrade, C++14, introduce new advances and features which make more affordable and easier the development of software for complex systems. Following this tenet we have designed and developed a component-based service-oriented C++ middleware, called ISE, for distributed systems using exclusively standard C++ and the quasi standard C++ Boost Libraries for keeping the middleware portable. The final aim of developing ISE has been to build the remote communication software infrastructure of an oceanic autonomous robotic sailboat called A-Tirma.

1 Introduction

The development of an autonomous sailboat is a demanding endeavor that is not circumscribed to the vessel. Normally, it implies also the development of a communication infrastructure which may consume a considerable amount of resources. The complexity of this communication infrastructure is commonly overlooked but it is - undoubtedly a central piece of the whole system, which may become quite complex in terms of software engineering. In this paper we introduce ISE, a component-based C++ middleware called ISE (Integrating Software Entities) we have developed specifically for building the communication software infrastructure for an autononomous sailboat. We have used a CBSE (Component-Based Software Engineering) paradigm [1] and a service-oriented architecture [2] to define a service-oriented component model. In ISE, a robotic system is a distributed system [3] built up of components, where each component provide and/or require services to/from others. Moreover, it is possible to dynamically reconfigure a whole system, as ISE provides support for on-demand dynamic component instantiation, and dynamic runtime type information about a system and its components. As case application in a real scenario, we have implemented using ISE first operative version of a distributed software infrastructure for remote control and monitoring of the autonomous sail vessel A-Tirma [4] developed at our laboratory. In the rest of the document we describe in more detail the middleware, how we have applied it so far to our projet A-Tirma, and the conclusions drawn from the work underdone.

† Instituto Universitario SIANI (www.roc.siani.es), Universidad de Las Palmas de Gran Canaria, Spain
‡ Departamento de Informática y Sistemas (www.dis.ulpgc.es), Universidad de Las Palmas de Gran Canaria, Spain
§ Correponding author's e-mail: antonio.dominguez@ulpgc.es

2 ISE middleware

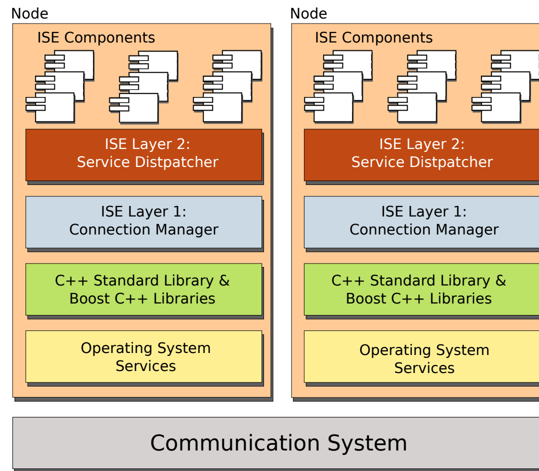

Fig. 1: ISE layered architecture

As main design principle for developing ISE we highlight the use of *standard and portable C++* as implementation language. C++ is a systems programming language that offers good use of hardware, effective abstraction and allow for real time performance [5]. With the appearance of new C++ standards (C++11 and C++14, implemented by most mainstream compilers [6]), the language has been modernized with many new features (lambda functions, move semantics, etc. [7]) which makes C++ code simpler. In addition to guarantee portability, we have also used Boost C++ libraries [8] given their “quasi-standard” status within the C++ community. Since software components, as composition units to integrate systems, provide a higher level of encapsulation, abstraction decoupling and reuse for software systems, a *component based* middleware [1][9][10] has been designed and developed. Moreover, the middleware is *service-oriented*, that is, component’s external interfaces will be defined as services provided and/or required [2].

In Fig. 1 we can appreciate the distributed layered architecture of ISE, organized in two layers. The first layer, *the connection manager*, is responsible for managing peer-to-peer connections amongst components, which isolates users and developers from the low level details for handling asynchronous connections using both TCP and UDP protocols. The second layer, *the service dispatcher*, abstracts the dispatching and housekeeping of all services provided by ISE components in order to carry out their specific functionalities. Those two layers support the runtime software infrastructure and the application programming interface (API) the middleware provides to ISE software components, as we can see in the figure.

Software components (components for short) are distributed *active objects* [11], in the sense that each of them has its own flow of execution, mapped as operating system processes or threads in a distributed system. In ISE, the external interface of a component is the enumeration of the services it provides (*provided* services), and also of the *potential* services it might use or require from others (*required* services). Therefore, any middleware interaction amongst components is abstracted by using services, and, to keep things simple, there are only two generic types of services. Namely, *request-response* and *subscription* services.

Request-response services. The sequence diagram of Fig. 2 illustrates how a request-response service operates. A service is started with a request addressed to

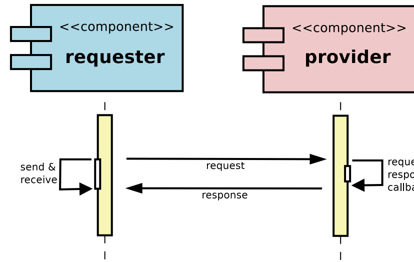

Fig. 2: Request-response service.

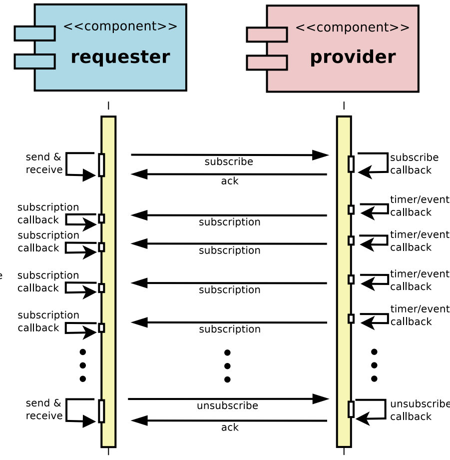

Fig. 3: Subscription service.

the provider by the requester. The provider answers with a response according to the request received. Although not shown in the figure for clarification reasons, in ISE, on the requester side, the component might do a blocking wait for the response. In the provider side service requests are received and processed asynchronously, usually associated to *callbacks* which are user-defined. This type of service put into practice a *pull* model of communication.

Subscription services. Fig. 3 depicts the sequence diagram of a typical subscription service in ISE [10]. It implements a publish/subscribe paradigm of communication. A service is initiated by the requester by sending a subscription request which. Once subscribed, the provider will start sending *subscriptions* (published data) to the requester, either periodically or in an asynchronous event basis. In normal conditions, the provider will keep serving subscriptions until the requester explicitly requests to get unsubscribed. Several callbacks are associated to both sides. On the requester side, to subscriptions when received. On the provider side, to requests for getting subscribed or unsubscribed, and to publish data periodically using timers or events. Subscription services provide a *push* model of communication.

To implement both types of generic services in ISE we have made use of Boost C++ library Boost.Asio [12], a library for network and low level I/O programming. In particular, ISE services component interactions are carried out sending packets through TCP connections or UDP datagrams between the components involved.

More specifically, in ISE, service interactions amongst components are done transferring discrete units of information we name *packets*. Furthermore, the services a component provides are defined by the type of packets each service uses. To provide generality and complete freedom of implementation to component developers, component's services and service type packets are all user-defined in ISE. Thus, for response-request services two types of packets define a service: a *request* packet and a *response* packet, shown in Fig. 2. Likewise, subscription services are

defined by five types of packets, namely: two packets for getting subscribed (request and response), a packet for *subscriptions*, and two packets for getting unsubscribed (request and response), all of them shown in Fig. 3.

```
ISE_DEFINE_PACKET_IN_NAMESPACE(
  ADD_NAMESPACE( a_tirma_tools ),
  subscription_rq,
  ADD_MEMBER( bool, status )
)
ISE_DEFINE_PACKET(
  subscription_rs,
  ADD_MEMBER( bool, status )
  ADD_MEMBER( std::string, ticket )
)
ISE_DEFINE_PACKET_IN_NAMESPACE(
  ADD_NAMESPACE( a_tirma_tools ),
  unsubscription_rq,
  ADD_MEMBER( std::string, ticket )
)
ISE_DEFINE_PACKET_IN_NAMESPACE(
  ADD_NAMESPACE( a_tirma_tools ),
  unsubscription_rs,
  ADD_MEMBER( bool, status )
)
ISE_DEFINE_PACKET_IN_NAMESPACE(
  ADD_NAMESPACE( a_tirma_tools ),
  xb_local_sb,
  ADD_MEMBER( int, power)
  ADD_MEMBER( int, temperature)
  ADD_MEMBER( int, rssi)
  ADD_MEMBER( int, failing_deliveries)
)
```

Fig. 4: C++ Packet definitions for a subscription service

```
namespace a_tirma_tools
{
  ...

  ISE_PROVIDED_SERVICES(
    a_tirma_xb_services_to_provide,

    ... // rest of services

    SUBSCRIPTION_PROVIDED_SERVICE(
      subscription_rq,
      subscription_rs,
      unsubscription_rq,
      unsubscription_rs,
      xb_local_sb,
      xb_local // service
    )
  )
}
```

Fig. 5: C++ service definition for a subscription service

ISE packets are easily defined directly in C++ code as Fig. 4 illustrates. Each type of packet is mapped as a C++ class. Packet data members can be defined to have any of the following types: C++ primitive types, `std::string`, any other ISE packet type and the container type `ise::packet::containers::vector`. As to packet's marshaling and unmarshaling the serialization interface for packets has been implemented using the Boost.Serialization Library [13]. Fig. 4 shows examples of packets used in a real system. Concretely, packets `subscription_rq`, `subscription_rs`, `unsubscription_rq` and `unsubscription_rs` are definitions shared by several distinct subscription services. Services are defined in a similar manner. In Fig. 5 we show an example of a subscription service, `xb_local`, defined with the packets of Fig. 4. ISE packets and services are usually defined in C++ header files, which may be included wherever necessary. The complete code behind ISE packets and services is generated using macros like the ones appearing in the previous figures, which have been implemented using the Boost.Preprocessor Library [14]. The use of this library has allowed to utilize C++ itself as IDL (Interface Definition Language) for defining interfaces between ISE components, in contraposition to other middlewares where usually IDL code must be compiled to generate its corresponding code in the final implementation language, (C, C++, Java, Python, etc.). We consider this an interesting simplification that facilitates middleware utilization and reduces its learning curve.

ISE provides a C++ class which embodies what is a component in the middleware, class `ise::component`. Using this class, either through derivation or through composition, we can define our own components endowed with its particular functionalities. As illustrated in Figures 2 and 3, in a given component, the functionality provided by a specific service is defined using *callbacks*, namely: function objects, lambdas, function pointers, etc. And it is through services that we define the behavior and functionality of an ISE component. Thence, for request-response services, in the provider side we associate a callback for the request packet. And similarly, for subscription services, in the provider side, we associate callbacks to requests for getting subscribed or unsubscribed. Accordingly, in the requester side, we associate a callback for the subscription packets it will receive when subscribed.

As general services, the middleware provides services for remote instantiation and name resolution. More concretely, it is possible to instantiate remotely ISE components using the services offered by a type of component provided by the middleware itself (component *instantiation_server*). We have used Boost C++ library Boost.Process [15] to implement ISE's instantiation services. In addition, the middleware also includes a type of component, called *name server*, which provides name resolution services. In fact, name servers allow to register information related to how to locate component instances, the services (and its packets) they provide, and, if any, about the instantiation servers present in a system as well. They allow also to define component namespaces, in order to organize logically a distributed system. Furthermore, name servers are also part of the infrastructure which allows to instantiate packets dynamically. Indeed, the Boost C++ library Boost.Any [16] has been used to implement the dynamic type safe behavior of ISE packets.

3 A-Tirma's Communication Infrastructure

We have designed and developed ISE for supporting the software communication infrastructure of an oceanic autonomous sailboat [4], developed at our laboratory, called *A-Tirma* (shown in Figures 6 and 7). Fig. 8 displays the deployment diagram of the embedded system on-board A-Tirma. The hardware of the main controller system is an Atmel ATSAM3X8E microcontroller based Arduino DUE prototyping platform. This main controller is in charge of sailboat autonomous navigation and short range communications via an XBee 868 Pro radio link[17]. The autonomous navigation control software it executes runs on ChibiOS [18] (more details in [19]). A second controller based on a Waspmote [20], an Atmel ATmega1281 microcontroller based prototyping board, is responsible for long-range communications and last resource localization signaling in case of emergency. The communcation controller acts as a gateway for sending and receiving, telemetry and remote telecommands. It uses a 3G/GPRS module[1] for data communications via the mobile phone network, and a Rockblock 9602 transceiver[2] for satellite communications utilizing

[1] Waspmote GPRS + GPS Module (`https://www.cooking-hacks.com/gprs-gps-module`).

[2] RockBLOCK Iridium SDB 9602 Transceiver (`http://www.rock7mobile.com/products-rockblock`).

Fig. 6: A-Tirma G2.

Fig. 7: A-Tirma in the water

the Iridium SBD (Short Burst Data) service [3]. XBee radio communications are used at short range (about 1-2 kms.) and real-time communication with the autonomous vessel. GPRS mobile communications have been included to communicate with the vessel at long range when situated in areas of mobile network communication coverage, typically coastal areas. Finally, for oceanic long range communication where mobile communications are not possible, the communication with the sailboat is based on Iridium SBD satellite communications. In addition to those three types of communications links, we can command directly the rudders and the sails of our vessel using a remote-controlled device (through an RC receiver). This link is wired at low level for direct control of the vessel's actuators and usually it is used when the sailboat is at sight. Mind, as well, that using GPRS and Iridium SBD links implies paying for the communications, whether the mobile telephone network fee, or the Iridium satellite access and data fees, being significantly more expensive the latter ones.

Fig. 9 depicts the deployment diagram for the communication infrastructure we have developed using ISE. Mind that the middleware has been not used on the sailboat, as the microcontroller platforms on-board are not computationally powerful enough, specially for their lack of memory. Indeed, we have selected them mainly for their low power consumption requirements ([21][4]). In fact, neither the main controller nor the communication controller have support for the C++ Standard Library, or the C++ Boost Libraries, which are necessary for running the middleware. The figure shows a typical system deployment in a navigation mission. As we can observe, in addition to the components provided by the middleware itself providing name and instantiation services, there are four types of nodes:the autonomous sailboat itself, a *short range communication* node, a *communication* node for long

[3] Iridium SBD (`https://www.iridium.com/services/details/iridium-sbd`).

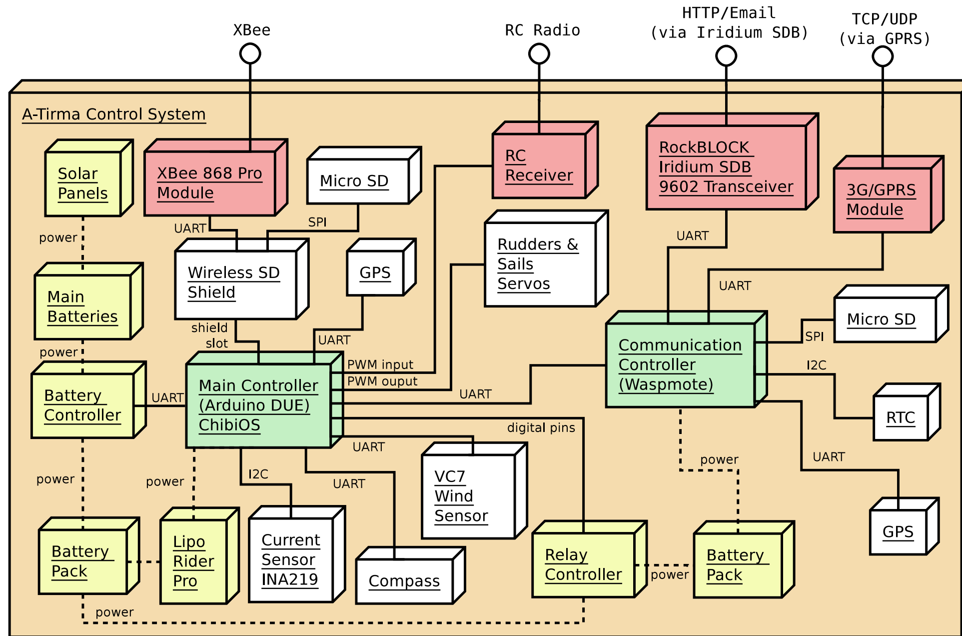

Fig. 8: A-Tirma's embedded control system. Deployment diagram. Power connections between elements are shown as discontinued lines, for clarification reasons not all of them are shown. For the same reasons, physical switches are also not shown in the diagram.

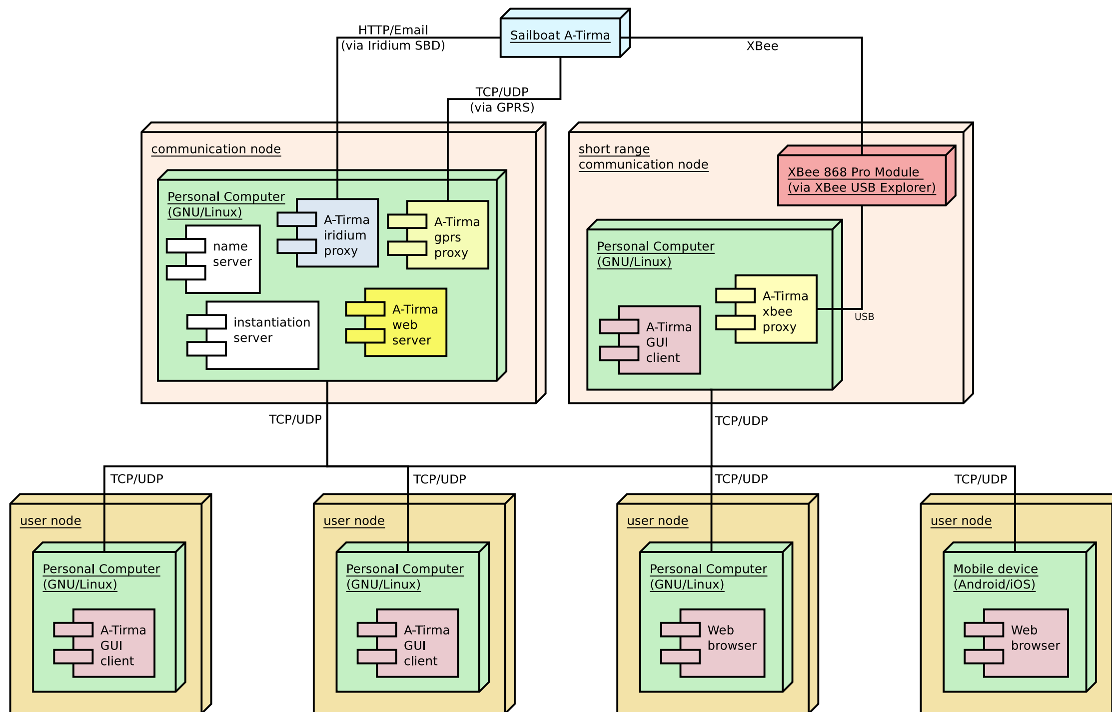

Fig. 9: A-Tirma's communication infrastructure. Deployment diagram. To keep the diagram as clear as possible, connections between ISE components are not shown. All ISE services are provided via TCP/UDP, so, implicitly, any component can access any service provided by any component.

range communications and several *user* nodes. Tables 1 and 2 describe briefly the ISE component and services building up the system. To keep Fig. 9 simple, ISE connections between component are not shown. Note that all ISE services are provided through TCP or UDP protocols (this is specified at component instantiation), components implicitly connect to their service providers as specified in Tables 1 and 2.

Services			
Service	**Type**	**Provided by**	**Description**
`telemetry`	subscription	XBee, GPRS and Iridium SBD proxy components	Provides periodically telemetry information to subscriber components. Subscription telemetry packets are sent to subscribers in raw format, as they come from the boat, subscribers are responsible for parsing telemetry packets. There are several telemetry types depending on the information they transport (location, waypoints, configuration parameters, etc.)
`telecommand`	request-response	XBee, GPRS and Iridium SBD proxy components	Service provided to send telecommands to the sailboat. Telecommand packets are also sent in raw format (the one accepted by the sailboat). There are also distinct types of telecommand packets which allow to control many aspects during a sailboat navigation mission.
`xb_local`	subscription	XBee proxy components	Provides periodically to subscribers information about the XBee link. The data provided by `xb_local` packets is in relation mainly to the quality of the signal with the boat (RSSI, XBee packet retries, etc.).
`xb_local_param`	request-response	XBee proxy components	Through this service is possible to change the power behavior (emission power) of the XBee link between the sailboat and the XBee proxy component involved.

Table 1: A-Tirma communication system. ISE services.

As we can observe in Fig. 9, usually a *short range communication* node is deployed in a typical mission scenario, specially if we will be supervising and monitoring the navigation of the boat at short distance using the XBee radio link. The hardware we usually utilize for deploying this node is a notebook under GNU/Linux Ubuntu, or a Raspberry PI under Raspbian exporting its graphical interface via VNC on a 10”-Android-based tablet. This node normally host a XBee proxy component and a GUI client component. The *communication* node typically is deployed on a personal computer under GNU/Linux Ubuntu in our lab, and usually host the proxy component we need for long range communications, namely the GPRS proxy component and the Iridium SBD proxy component. In addition, it ordinarily hosts also a web server component which uses the services of all proxy communication components (XBee, GPRS and Iridium SBD ones) for publishing the sailboat’s tracking telemetry information through a web page. As to the *user* nodes appearing in Fig. 9, two of them host mainly a GUI client component instance. Those components use the services provided by the proxy communication components running in the communication nodes and implement a remote interface for monitoring and control of the sailboat. A snapshot of the interface provided by GUI client component instances is shown in Fig. 10. They may be instantiated at different user nodes, so several users may be remotely controlling and monitoring the vessel at any given

Components			
Component	**Services provided**	**Services used**	**Description**
XBee proxy component	`telemetry`, `telecommand`, `xb_local` and `xb_local_params`	none	It is a proxy component to communicate with the sailboat through an XBee 868 Pro radio link. Allows other components to get subscribed for telemetry information, and accepts telecommands addressed to the boat by requester components.
GPRS proxy component	`telemetry` and `telecommand`	none	Proxy component for communication with the boat through a GPRS data link via TCP/UDP protocols. Allows other components to get subscribed for telemetry information, and accepts telecommands addressed to the boat by requester components.
Iridium SBD proxy component	`telemetry` and `telecommand`	none	Proxy component which communicates with the sailboat through the Iridium Short Burst Data (SDB) service via HTTP and email protocols. Allows other components to get subscribed for telemetry information, and accepts telecommands addressed to the boat by requester components.
GUI client component	none	`telemetry`, `telecommand`, `xb_local` and `xb_local_params`	This component does not provides any ISE services, and can use any of the services provided by XBee, GPRS and Iridium SBD proxy components. It is a subscriber and requester of telemetry and telecommand (and `xb_local` and `xb_local_param`) services, respectively.
Web server component	none	`telemetry`	This component is a subscriber of telemetry services, and provides a web page service with A-Tirma's tracking and telemetry information during a mission.
Name server	ISE naming services		Component which provides ISE naming services. Generic component provided by the middleware.
Instantiation server	ISE remote instantiation services		This is a generic component provided by the middleware itself. It provides ISE remote instantiation services.

Table 2: A-Tirma communication system. ISE components.

moment. Using them we can monitor and control many aspects of the embedded system onboard vessel A-Tirma. Thus, for example, it is possible to modify parameters affecting the sailboat behavior during autonomous navigation like time to next tacking, forbidden angle range to navigate up and down wind, period of telemetry messages, whether activating leeway compensation or not, adding/removing navigation waypoints, changing vessel's navigation mode, etc. just to name a few. The hardware we have ordinarily use for running user nodes is a typical personal computer under GNU/Linux Ubuntu. Furthermore, using the web service provided by the web server component hosted in the communication node, we can access a tracking web page during a mission, using a typical personal computer, or a mobile device like a smartphone or tablet (Fig. 11 shows a snapshop of this tracking web page).

In order to test the communication infrastructure, amongst the different real on-field tests we have carried out, in Fig. 12, we provide the tracking log of a 10-hour stress test which took place in the bay of El Confital in Gran Canaria. In this stress test A-Tirma kept navigating autonomously during about 9-10 hours of continuous navigation following consecutively two sailing triangles. The objective of the test was to verify the robustness of the embedded autonomous system on-board and of the distributed communication software infrastructure built using ISE. Concretely,

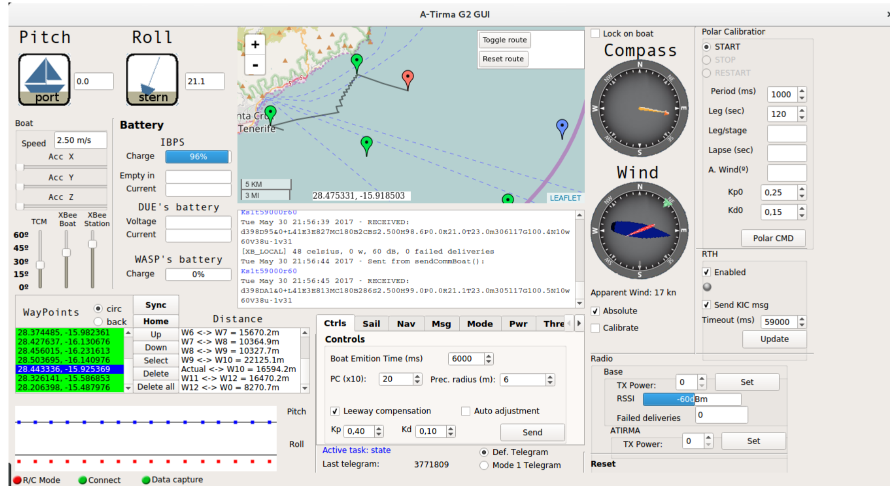

Fig. 10: A-Tirma's control and monitoring interface

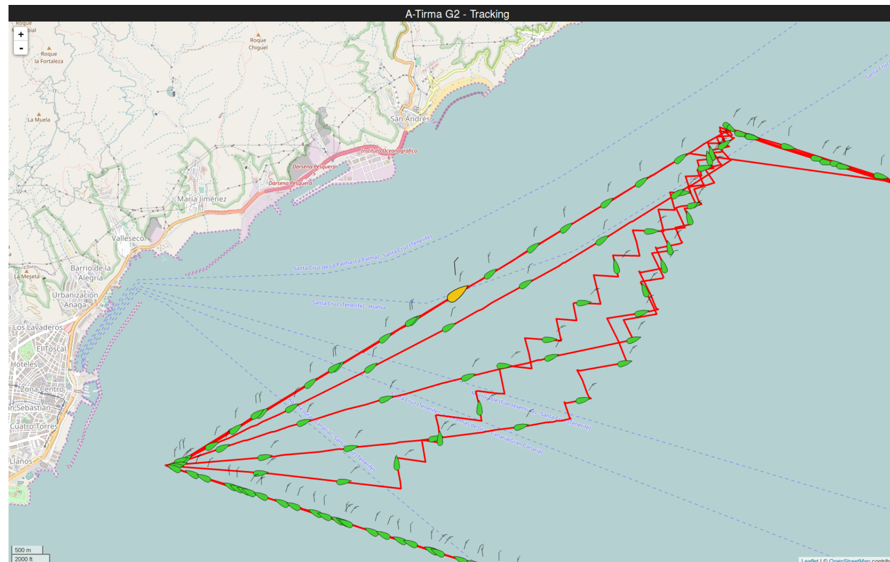

Fig. 11: A-Tirma's web tracking web page.

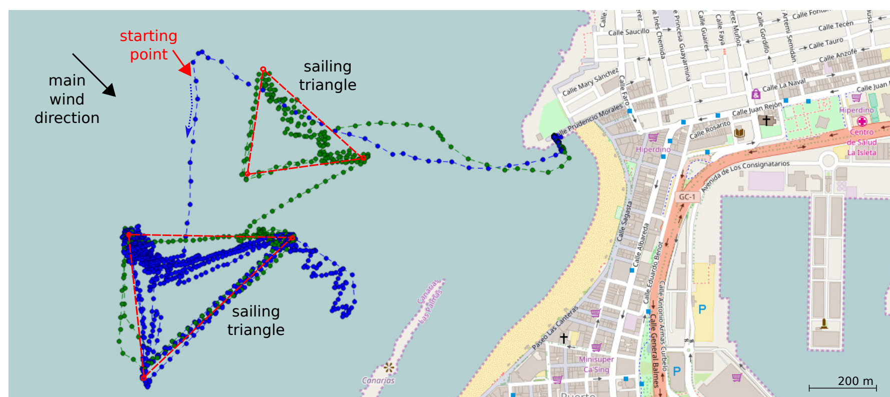

Fig. 12: A-Tirma's 10-hour stress test tracking log.

in this 10-hour stress test, we deployed the system using a *short range comunication* node, and two *communication* nodes running remotely in different machines in our laboratory, where one of them was just a secondary node for backing the main one in case it failed. As to *user* nodes, we deployed on-field two nodes running the GUI component of Fig. 10 in two different notebooks under GNU/Linux.

4 Conclusions

In this paper we have outlined the design and development of a distributed component based and service-oriented C++ middleware called ISE, designed and developed to build the communication software infrastructure of an oceanic autonomous sailing vessel called *A-Tirma*. One of the main objective of developing ISE was to try to take advantage of the new C++ standard, C++11, and its upgrade, C++14, for developing software for distributed complex systems. Furthermore, we have used exclusively standard C++ and the highly portable and quasi standard Boost C++ Libraries to keep the middleware portable, taking advantage of the important metaprogramming features available in this new standards. As an interesting point of simplification of use, we have made an important effort to avoid the use of an specific Interface Definition Language (IDL), in ISE the IDL is C++ itself. This is quite common in other tools for developing distributed software, as CORBA [22], in general, and ROS [23] in the particular area of robotics, as paradigmatic examples. Finally, we have deployed and tested this infrastructure in a real experimental setup to assess its continuous operation during a significant interval of time in terms of system robustness and reliability with satisfactory results.

Acknowledgements

The authors are sincerely grateful to Solumatica Canarias for providing financial support for building the A-TIRMA G2 prototype, and to the Real Club Náutico de Gran Canaria and to the Real Club Victoria for the access they granted to their facilities, and for the logistical support during the development of this project.

References

1. T. Vale, I. Crnkovic, E. S. de Almeida, P. A. da Mota Silveira Neto, Y. C. Cavalcanti, and S. R. de Lemos Meira, “Twenty-eight years of component-based software engineering,” *Journal of Systems and Software*, vol. 111, pp. 128 – 148, 2016. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S0164121215002095
2. L. B. R. Oliveira, F. S. Osório, and E. Y. Nakagawa, “An investigation into the development of service-oriented robotic systems,” in *Proceedings of the 28th Annual ACM Symposium on Applied Computing*, ser. SAC ’13. New York, NY, USA: ACM, 2013, pp. 223–228. [Online]. Available: http://doi.acm.org/10.1145/2480362.2480410

3. G. J. D. T. K. G. B. Coulouris, *Distributed Systems: Concepts and Design (5th Edition)*, 5th ed. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 2011.
4. A. C. Domínguez-Brito, B. Valle-Fernández, J. Cabrera-Gámez, A. Ramos-de Miguel, and J. C. García, *A-TIRMA G2: An Oceanic Autonomous Sailboat*. Cham: Springer International Publishing, 2016, ch. A-TIRMA G2: An Oceanic Autonomous Sailboat, pp. 3–13. [Online]. Available: http://dx.doi.org/10.1007/978-3-319-23335-2_1
5. B. Stroustrup, *Abstraction and the C++ Machine Model*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2005, pp. 1–13. [Online]. Available: http://dx.doi.org/10.1007/11535409_1
6. isocpp.org. (2018) C++ Super FAQ. When will compilers implement C++14? [Online]. Available: https://isocpp.org/wiki/faq/cpp14#cpp14-compilers
7. B. Stroustrup, *The C++ Programming Language, 4th Edition*. Addison-Wesley Professional, 2013. [Online]. Available: http://www.stroustrup.com/4th.html
8. Boost.org. (2018) Boost C++ Libraries. [Online]. Available: http://www.boost.org
9. D. Brugali and P. Scandurra, “Component-based robotic engineering (part i) [tutorial],” *IEEE Robotics Automation Magazine*, vol. 16, no. 4, pp. 84–96, December 2009. [Online]. Available: http://www.doi.org/10.1109/MRA.2009.934837
10. D. Brugali and A. Shakhimardanov, “Component-based robotic engineering (part ii),” *IEEE Robotics Automation Magazine*, vol. 17, no. 1, pp. 100–112, March 2010. [Online]. Available: http://www.doi.org/10.1109/MRA.2010.935798
11. C. Ellis and S. Gibbs, *Object-Oriented Concepts, Databases, and Applications*. ACM Press, Addison-Wesley, 1989, ch. Active Objects: Realities and Possibilities.
12. C. Kohlhoff. (2018) Boost.Asio. Boost C++ Libraries. [Online]. Available: http://www.boost.org/doc/libs/release/libs/asio/
13. R. Ramey. (2018) The Boost Serialization Library. Boost C++ Libraries. [Online]. Available: http://www.boost.org/doc/libs/release/libs/serialization/
14. V. Karvonen and P. Mensonides. The Boost Preprocessor Library. Boost C++ Libraries.
15. K. D. Morgenstern. (2017) The Boost Process Library. Boost C++ Libraries. [Online]. Available: http://www.boost.org/doc/libs/release/libs/process/
16. K. Henney. (2017) The Boost.Any Library. Boost C++ Libraries. [Online]. Available: http://www.boost.org/doc/libs/release/libs/any
17. Digi. (2018) XBee/XBee Pro. 868 RF Module. User Guide. [Online]. Available: https://www.digi.com/resources/documentation/digidocs/pdfs/90001020.pdf
18. Giovanni Di Sirio. (2017) ChibiOS free embedded RTOS. [Online]. Available: http://www.chibios.org
19. J. Cabrera-Gámez, A. Ramos-de Miguel, A. C. Domínguez-Brito, J. D. Hernández-Sosa, J. Isern-González, and L. Adler, “A real-time sailboat controller based on chibios,” in *Proceedings of the 7th International Robotic Sailing Conference. Robotic Sailing 2014*, F. Morgan and D. Tynan, Eds. Springer International Publishing, 2014, pp. 77–85. [Online]. Available: http://dx.doi.org/10.1007/978-3-319-10076-0_7
20. Libelium. (2017) Waspmote - Open Source Sensor Node for the Internet of Things. [Online]. Available: http://www.libelium.com/products/waspmote
21. J. Cabrera-Gámez, A. Ramos-de Miguel, A. Domínguez-Brito, J. Hernández-Sosa, J. Isern-González, and E. Fernández-Perdomo, “An embedded low-power control system for autonomous sailboats,” in *Proceedings of the 6th International Robotic Sailing Conference. Robotic Sailing 2013*, F. L. Bars and L. Jaulin, Eds. Springer International Publishing, 2013, pp. 67–79. [Online]. Available: http://dx.doi.org/10.1007/978-3-319-02276-5_6
22. M. Henning and S. Vinoski, *Advanced CORBA Programming with C++*, ser. Addison-Wesley Professional Computing Series. Addison-Wesley, 1999.
23. M. Quigley, K. Conley, B. Gerkey, J. Faust, T. Foote, J. Leibs, R. Wheeler, and A. Y. Ng, “Ros: an open-source robot operating system,” in *ICRA workshop on open source software*, vol. 3, no. 3.2, 2009, p. 5. [Online]. Available: http://www.willowgarage.com/sites/default/files/icraoss09-ROS.pdf